

# Construction and Verification of Software

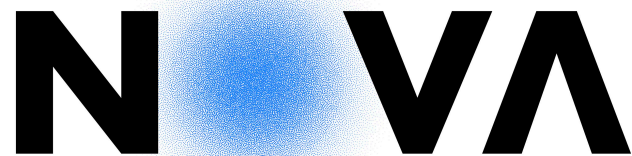
2021 - 2022

**MIEI - Integrated Master in Computer Science and Informatics**  
Consolidation block

**Lecture 9 - Concurrent Abstract Data Types**

**Bernardo Toninho** ([btoninho@fct.unl.pt](mailto:btoninho@fct.unl.pt))

based on previous editions by **João Seco** and **Luís Caires**



# Outline

---

- Concurrent ADTs
- Concurrency Control (Monitors)
- Verifying Monitors
- Applying and Verifying Monitor Conditions
- Concurrent ADTs construction
- An example of construction

# Part I

# Concurrent ADTs

# Concurrency

---

- To execute several units of computation out of the prescribed order (non strictly sequential).
- An undetermined number of interleaving executions between units
- Allows parallel computation (Hardware permitting) - no state sharing
- Allows asynchronous programming (blocking IO)
  - improves efficiency
  - improves responsiveness (UI)
- Examples of models with concurrency:
  - Message passing
  - Shared Memory
  - Optimistic (Transactions)

# Concurrency

---

- To execute several units of computation out of the prescribed order (non strictly sequential).
- An undetermined number of interleaving executions between units
- Allows parallel computation (Hardware permitting) - no state sharing
- Allows asynchronous programming (blocking IO)
  - improves efficiency
  - improves responsiveness (UI)
- Examples of models with concurrency:
  - Message passing
  - **Shared Memory <<< most common in the imperative setting**
  - Optimistic (Transactions)

# Concurrency with shared memory

---

- Several threads of execution **share** the same state footprint
- **Interference** is expected:
  - the local view of a thread may change without notice (another thread may act “under the hood” and cause state changes).
- Interference is **the essence** of concurrency
- **Key issue:**
  - how to keep state consistency in the presence of state sharing and control interference
  - how to reason about the effects of concurrent code
- Reasoning about concurrency is very **challenging!**
- Modularity brought by ADT based design is crucial!

# Verification of ADTs

---

- The verification of sequential uses of ADTs characterizes the observable states of objects implementing them.
- All ADT operations (**public** methods) preserve the consistency of the ADT.
- Consistency is expressed by the representation and abstract invariants (and the abstraction mapping).
- The following Hoare/Sep Logic triple is valid for all method bodies (**mbody**) of ADT operations
$$\{ \text{RepInv} \ \&\& \ \text{pre-cond} \} \ \text{mbody} \ \{ \text{RepInv} \ \&\& \ \text{post-cond} \}$$
- This line of reasoning works well under the assumption of sequentiality. What if method executions overlap in time?

# Verification of Concurrent ADTs

---

- **Challenge:** how to program and reason about ADTs with interfering methods



# Interference between ADT operations

---

- Consider a Stack ADT
  - **push(v), pop(), isEmpty()**
  - push() interferes with pop() ?
  - pop() interferes with isEmpty() ?
  - pop() interferes with pop() ?
- Consider a Dictionary ADT
  - **assoc(key,data), find(key)**
  - assoc() interferes with find() ?
  - assoc() interferes with assoc() ?
  - find() interferes with find() ?

# Verification of Concurrent ADTs

---

- **Challenge:** how to program and reason about ADTs with interfering methods
- The verification of concurrent uses of ADTs should also observe stable states of objects implementing them.
- All ADT operations (public methods) must also preserve the consistency of the ADT.
- The following Hoare/Sep Logic triple should also be valid for all method bodies (**mbody**) of ADT operations
$$\{ \text{RepInv} \ \&\& \ \text{pre-cond} \} \ \text{mbody} \ \{ \text{RepInv} \ \&\& \ \text{post-cond} \}$$
- A sound approach is to consider the ADT operations as the unit of concurrency and structuring of reasoning (later to be refined).

# Operation Level Behaviour

---

- Reason at the level of ADT operations and not at the level of unstructured low level instructions and state changes
- Each ADT operation is performed in three steps
  - The operation is called (by the client thread)
  - The operation is executed (inside the ADT)
  - The operation returns
- Example:
  - push\_call(2)
    - ... **execute** (internally to the ADT)
  - push\_return
  - pop\_call()
    - .... **execute** (internally to the ADT)
  - pop\_return(2)

# Operation Level Behaviour

---

- We may consider several levels of concurrency
  - Several threads are invoking ADT operations but only one may actually be executing the operation
    - strict serialisation, easier to implement and reason about
    - less chances of “unsound” interference
  - Several threads are invoking ADT operations but more than one may be executing an operation
    - more parallelism, more concurrency, harder to implement and reason about
    - more chances of “unsound”/ bad interference
- How does the concurrent object behaviour relate to the intended sequential object specification ?

# Two Basic Models

---

- Serializability
  - The global trace is always consistent with some sequential serialisation of previous operations (**no overlaps** of calls and returns), compatible with the sequential specification.
- Linearizability
  - The global trace is always consistent with a view in which previous operations appear to occur instantaneously between calls and returns, and the obtained serialisation is compatible with the sequential specification.
- Linearizability is more flexible than serializability, as it allows for more parallel behaviour.

# Desired properties ADT operations (ACID)

---

- Atomicity
  - No intermediate states are visible  
(not compatible with the representation invariant)
- Consistency
  - Operations lead from a sound state to a sound state  
(invariants and soundness are preserved)
- Isolation
  - This is another word for “no unsafe interference”
- Durability (this goes without saying)
  - Effects are undoable (N.B: this is more useful to highlight in the context of database transactions)

# Correctness of Concurrent ADTs

---

- With naive concurrency, it is hard (or impossible) for client code to be sure if a specific **post-condition** holds.
- E.g: two clients modify the concrete state at the same time, bringing the state inconsistent, breaking the representation invariant, or even crashing the code.

```
void push(int v)
//@ requires StackInv(this,?n,?m) &*& n < m;
//@ ensures StackInv(this,n+1,m);
{
    int i = nelems;
    store[i] = v;
    nelems++;
}
```

```
// Thread 1           // Thread 2
int i = nelems;
store[i] = v;
nelems = i + 1;

↓

int i = nelems;
store[i] = v;
nelems = i + 1;
```

# Correctness of Concurrent ADTs

---

- With naive concurrency, it is hard (or impossible) for client code to be sure if a specific **post-condition** holds.
- E.g: two clients modify the concrete state at the same time, bringing the state inconsistent, breaking the representation invariant, or even crashing the code.
- Solution using serialisation:
  - **serialise** usages of concrete states, so that just a **single** thread may be accessing the state at each given moment (mutual exclusion of concrete state)
  - We may then safely reason about such mutually exclusive code fragments as we have done for sequential code.



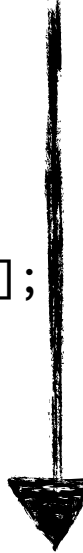
# Correctness of Concurrent ADTs

- With naive concurrency, it is hard (or impossible) for client code to be sure if a specific **post-condition** holds.
- E.g: two clients modify the concrete state at the same time, bringing the state inconsistent, breaking the representation invariant, or even crashing the code.

```
int pop()
/*@ requires StackInv(this,?n,?m) &*& n > 0;
   @ ensures StackInv(this,n-1,m);
{
    int v = store[nelems-1];
    nelems--;
    return v;
}
```

```
           // Thread 1
pop() {
    int v = store[nelems-1];
    nelems--;
    return v;
}

           // Thread 2
pop() {
    int v = store[nelems-1];
    nelems--;
    return v;
}
```

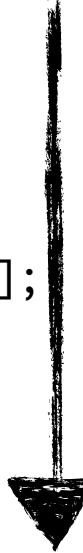


# Correctness of Concurrent ADTs

- With naive concurrency, it is hard (or impossible) for client code to be sure if a specific **post-condition** holds.
- E.g: two clients modify the concrete state at the same time, bringing the state inconsistent, breaking the representation invariant, or even crashing the code.

```
int pop()
/*@ requires StackInv(this,?n,?m) &*& n > 0;
   *@ ensures StackInv(this,n-1,m);
{
    int v = store[nelems-1];
    nelems--;
    return v;
}
```

```
           // Thread 1
pop() {
           // Thread 2
pop() {
    int v = store[nelems-1];
    nelems--;
    return v;
}
}
return v;
}
```



Is this safe in all situations?

# Correctness of Concurrent ADTs

---

- With naive concurrency, it is hard (or impossible) for client code to be sure if a specific **pre-condition** holds.
- E.g: client checks that a buffer is not empty, but other thread empties it under the hood.
- Solution:
  - Concurrency control replaces pre-condition checking (on the client side) by explicit waiting for the precondition to hold (inside the ADT).
  - The pre-condition for some ADT op can only be enabled by executing some other ADT op
  - So waiting for a pre-condition must be managed by special programming language or system support, in a coordinated way with other ADT operations

# Concurrent Programming

---

- Reasoning about concurrency is **hard**
- Making sure the code is right is much more difficult than in sequential code
- Trying to simulate the program running in your head and debugging it does not really work anymore :-)
  - It does not really work in the sequential case either, actually..., although you may still believe.
- We will now study how to design and construct correct concurrent code, based on **monitors**
- Monitor = invariant preserving concurrent ADT
- Nicely supported by `java.concurrent.util`

Part II

Concurrency Control via  
Monitors

# Monitors

Operating  
Systems

C. Weissman  
Editor

## Monitors: An Operating System Structuring Concept

C.A.R. Hoare  
The Queen's University of Belfast

**This paper develops Brinch-Hansen's concept of a monitor as a method of structuring an operating system. It introduces a form of synchronization, describes a possible method of implementation in terms of semaphores and gives a suitable proof rule. Illustrative examples include a single resource scheduler, a bounded buffer, an alarm clock, a buffer pool, a disk head optimizer, and a version of the problem of readers and writers.**

**Key Words and Phrases:** monitors, operating systems, scheduling, mutual exclusion, synchronization, system implementation languages, structured multiprogramming  
**CR Categories:** 4.31, 4.22



# Monitors

---

- An ADT where operations may be called concurrently
- **2 key mechanisms** provided for ensuring consistency:

## **Synchronization** (a.k.a. mutual exclusion)

- only a single thread may “own” the shared state at any time object, and has permission to change it
- All that client code may expect from shared state is **the invariant, and nothing more than the invariant**
- any context switches must preserve the invariant (observable states)

## **Concurrency control**

- pre-condition checking must be usually replaced by explicit waiting for the pre-condition to hold.
- conditions refine the invariant into finer partitions.



# Implementation of monitors

---

- To implement monitors in Java, we will use locks
  - You have already heard about locks (FSO, CP)
  - A lot harder to reason about programs if we just think of using locks in an unstructured way
- We may later refine the borders of serialisability to get more concurrency (approach linearisability)
  - Use locks as delimiters of abstract operations on the shared state
  - Use the **java.util.concurrent** API (Doug Lea)
  - Will learn how to design concurrent ADTs without thinking “operationally”, but rather in terms of (partitioned) ownership, invariants, and conditions.



# Example (Bounded Counter)

---

- Consider a counter with a maximum value (Bounded)

```
class BCounter {  
    int N;  
    int MAX;  
    BCounter(int max) { N = 0 ; MAX = max; }  
    void inc() { N++; }  
    void dec() { N--; }  
    int get() { return N; }  
}
```

# Example (Bounded Counter)

---

```
/*@
  predicate BCounterInv(BCounter c; int v,int m) = c.N l-> v &*& c.MAX l-> m &*& v>=0 &*& v<=m;
  @*/
class BCounter {
  int N; int MAX;

  BCounter(int max)
  // @ requires 0 <= max;
  // @ ensures BCounterInv(this,0,max);
  { N = 0; MAX = max; }

  void inc()
  // @ requires BCounterInv(this,?n,?m) &*& n < m;
  // @ ensures BCounterInv(this,n+1,m);
  { N++; }

  void dec()
  // @ requires BCounterInv(this,?n,?m) &*& n > 0;
  // @ ensures BCounterInv(this,n-1,m);
  { N--; }

  int get()
  // @ requires BCounterInv(this,?n,?m);
  // @ ensures BCounterInv(this,n,m) &*& 0<=result &*& result<=m;
  { return N;}
}
```

# Example (Bounded Counter)

---

- The use of the bounded counter is safe in a sequential setting.

```
public static void main(String[] args)
//@ requires true;
//@ ensures true;
{
    int MAX = 100;
    BCounter c = new BCounter(MAX);
    //@ assert BCounterInv(c,0,MAX);
    if (c.get() < MAX) {
        c.inc(); // this is ok, precondition satisfied
    }
}
```

# Example (Bounded Counter)

---

- But... in a concurrent setting where the reference to the counter is used elsewhere??

```
public static void main(String[] args)
//@ requires true;
//@ ensures true;
{
    int MAX = 100;
    BCounter c = new BCounter(MAX);
    //@ assert BCounterInv(c,0,MAX);
    giveAway(c); // potentially give other thread access to c
    if (c.get() < MAX) {
        //@ assert BCounterInv(c,?v,MAX) &*& v < MAX;
        c.inc();
        // not safe any more as other thread may have acted
    }
}
```

# 1st: Serialise access to shared state

---

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
```

```
class CCounter {
```

```
    int N;
```

```
    int MAX;
```

```
    ReentrantLock mon;
```

```
    CCounter(int max) {
```

```
        N = 0;
```

```
        MAX = max;
```

```
        mon = new ReentrantLock();
```

```
    }
```

```
    ...
```

# Example (Bounded Counter)

---

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;
    ...
    void inc()
    {
        mon."enter"(); //request permission to the shared state
        N++;
        mon."leave"(); //release ownership of the shared state
    }

    void dec()
    {
        mon."enter"(); //request permission to the shared state
        N--;
        mon."leave"(); //release ownership of the shared state
    }
}
```

# Example (Bounded Counter)

---

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;
    ...
    void inc()
    {
        mon.lock(); //request permission to the shared state
        N++;
        mon.unlock(); //release ownership of the shared state
    }

    void dec()
    {
        mon.lock(); //request permission to the shared state
        N--;
        mon.unlock(); //release ownership of the shared state
    }
}
```

# Example (Bounded Counter)

---

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
```

```
class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;
```

...

```
int get()
{
    int r;
    mon."enter"();
    r = N; // put a copy on the stack, private to the thread
    mon."leave"();
    return r;
}
```



# Summary

---

- Undisciplined concurrent access to shared memory causes unexpected (thus invalid) behaviour.
- (Conservatively) Implement all operations in a monitor, where only one thread is allowed at a time.

```
T op()
{
    T result;
    mon."enter"();
    // Thread has access to shared state
    result = Expression;
    // Thread releases access to shared state
    mon."leave"();
    return result;
}
```

- (Next) will define more flexible mechanisms that allow more interleaving of operations.

# Part III

## Verifying Monitors

# Reasoning about monitor operations

---

- Monitor “guards” the access to the ADT shared state.  
*shared state = footprint* of the monitor operations
- The “*enter*” operation gives access to the footprint
- The “*leave*” operation captures the footprint back

# Hoare Rule for enter/leave (Lock/unlock)

---

$$\{ \text{emp} \} m.\text{enter}() \{ \text{SharedStateInv}() \}$$
$$\{ \text{SharedStateInv}() \} m.\text{leave}() \{ \text{emp} \}$$

**SharedStateInv()** is the representation invariant of the ADT that is used to verify operations in exclusive ownership of the monitor

In our example ...

```
//@ predicate CCounterInv(CCounter c) =  
    c.N |-> ?v &*& c.MAX |-> ?m &*& v >= 0 &*& v <= m;
```

so:

$$\{ \text{emp} \} m.\text{lock}() \{ \text{CCounterInv}(\text{this}) \}$$

# Hoare Rule for enter/leave (Lock/unLock)

---

- Representation Invariant is available inside the monitor

```
class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;

    void inc()
        //@ requires ...
        //@ ensures ...
    {
        //@ request permission to the shared state
        mon.lock();
        //@ assert CCounterInv(this,?n,?m)
        N++;
        //@ assert CCounterInv(this,n+1,m)
        mon.unlock();
        //@ release ownership of the shared state
    }
}
```

# Warning: Red assertions not available!

---

- Representation Invariant is available inside the monitor

```
class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;

    void inc()
        //@ requires CCounterInv(this,?n,?m) &*& n < m;
        //@ ensures  CCounterInv(this,n+1,m);
    {
        //@ request permission to the shared state
        mon.lock();
        //@ assert CCounterInv(this,?n,?m)
        N++;
        //@ assert CCounterInv(this,n+1,m)
        mon.unlock();
        //@ release ownership of the shared state
    }
}
```

# How can a client check $n < m$ ?

---

# How can an ADT capture the footprint in a lock?

---



# “invisible” abstract state

---

- Many threads may be interfering, so the only thing one may assume is the invariant, only after entering the shared state a client may know extra details about the concrete state.
- In fact, nothing specific about the abstract state may be revealed to client code, and we need to be less informative about the abstract state (e.g., no current val)
- Inside the object, the only unprotected objects are the locks (or the single lock).
- Each lock can be used to ask permission to access a disjoint part of the shared state.
- We must precisely define which part of the shared state is separately owned by each lock.

# Example (Bounded Counter)

```
/*@
    predicate_ctor CCounter_shared_state (BCounter c) () =
        c.N |-> ?v &*& v >= 0 &*& c.MAX |-> ?m &*& 0 < m &*& v <= m;
*/
/*@
    predicate CCounterInv(CCounter c) =
        c.mon |-> ?l
        &*& l != null
        &*& lck(l, 1, CCounter_shared_state(c))
*/
class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;

    CCounter(int max)
    //@ ensures CCounterInv(this);
    {
        N = 0 ;
        MAX = max;
        mon = new ReentrantLock();
    }
}
```

# First-order predicates

---

- Define a family of predicates that can be stored by other predicates through a predicate constructor

```
predicate_ctor CCounter_shared_state (CCounter c) () =  
  c.N |-> ?v &*& v >= 0 &*& c.MAX |-> ?m &*& 0 < m &*& v <= m;
```

- and then instantiate them to produce a predicate instance

```
lck(1, 1, CCounter_shared_state(c))
```

- and use it in other predicates

```
CCounter_shared_state(this)()
```

# ReentrantLock constructor

---

- The ReentrantLock class signature ensures the establishment of the “native” predicate  $\mathbf{lck}$  based on the invariant predicate  $\mathbf{inv}$

```
predicate enter_lck(real p, predicate() inv) = inv() ;  
predicate lck(Lock s; real p, predicate() inv);
```

```
public class ReentrantLock {  
    public ReentrantLock();  
    //@ requires enter_lck(1,?inv);  
    //@ ensures lck(this, 1, inv);  
    ...  
}
```

# Example (Bounded Counter)

---

```
class CCounter {
  int N;
  int MAX;
  ReentrantLock mon;
  CCounter(int max)
  //@ ensures CCounterInv(this);
  {
    N = 0 ;
    MAX = max;
    //@ close CCounter_shared_state(this);
    //@ close enter_lck(1, CCounter_shared_state(this));
    mon = new ReentrantLock();
    //@ assert lck(mon, 1, CCounter_shared_state(this));
    //@ close CCounterInv(this);
  }
}
```

# Example (Bounded Counter)

---

- Representation invariant captures the resources that can be used in a concurrent context

```
class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;

    void inc()
    //@ requires CCounterInv(this);
    //@ ensures  CCounterInv(this);
    {
        mon.lock(); // request permission to the shared state
        //@ open CCounter_shared_state(this)();
        N++;
        //@ close CCounter_shared_state(this)();
        mon.unlock(); // release ownership of the shared state
    }
}
```

# ReentrantLock operations

---

- The ReentrantLock methods `lock` and `unlock` use the predicate `lck` to make the invariant available inside the monitor and captured again outside the exclusive access region.

```
predicate lck(Lock s; real p, predicate() inv);
```

```
public class ReentrantLock {  
    ...  
    public void lock();  
    //@ requires lck(?t, 1, ?inv);  
    //@ ensures lck(t, 0, inv) &*& inv();  
  
    public void unlock();  
    //@ requires lck(?t, 0, ?inv) &*& inv();  
    //@ ensures lck(t, 1, inv);  
    ...  
}
```

# Example (Bounded Counter)

---

- Representation invariant captures the resources that can be used in a concurrent context

```
class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;

    void dec()
    //@ requires CCounterInv(this);
    //@ ensures CCounterInv(this);
    {
        mon.lock();
        //@ open CCounter_shared_state(this)();
        N--;
        //@ close CCounter_shared_state(this)();
        mon.unlock();
    }
}
```



# What if $N == \emptyset$ ?

---

# What if $N \geq 0$ ?

---

- The “public” representation invariant cannot reveal information to make the pre-condition hold.

```
class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;

    void dec()
    //@ requires CCounterInv(this); // no way to reveal a pre-cond!
    //@ ensures  CCounterInv(this);
    {
        mon.enter();
        //@ open CCounter_shared_state(this)();
        N--;
        //@ close CCounter_shared_state(this)(); // must ensure  $N \geq 0$ !
        mon.leave();
    }
}
```

# Summary

---

- This structured monitor implementation makes the shared state representation invariant available inside the exclusive access area.
- The operations **lock** and **unlock** release and capture the representation invariant.
- Operations **cannot** enforce preconditions in the **caller context** due to possible interleaving of operations outside the monitor.

# Part IV

# Conditions

# How can a client check $n < m$ ?

---

# What if $N == \emptyset$ ?

---

# How can a client check $n < m$ ?

---

- Concurrency control mechanisms replace pre-condition checking (on the client side) by explicit waiting for the precondition to hold (inside the ADT).
- The pre-condition for some ADT operation can only be enabled by executing some other ADT operation.
- Waiting for a pre-condition is managed by special programming language or system support, in a coordinated way with other ADT operations.

## Monitor Conditions

# Partition shared state using conditions

---

- Conditions implement queues of suspended threads...

```
class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;
    Condition notzero;
    Condition notmax;

    void dec()
    //@ requires CCounterInv(this);
    //@ ensures CCounterInv(this);
    {
        mon.enter();
        //@ open CCounter_shared_state(this)();
        if (N==0) notzero.await();
        N--;
        //@ close CCounter_shared_state(this)();
        mon.leave();
    }
}
```



# Partition shared state using conditions

---

- Conditions represent operations' preconditions, that are checked in the callee context, where access is granted

```
class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;
    Condition notzero;
    Condition notmax;

    void dec()
    //@ requires CCounterInv(this);
    //@ ensures CCounterInv(this);
    {
        mon.enter();
        //@ open CCounter_shared_state(this)();
        if (N==0) notzero.await();
        N--;
        //@ close CCounter_shared_state(this)();
        mon.leave();
    }
    ...
}
```

# Hoare Rule for *wait* (*await*)

---

$$\{ \text{SharedStateInv}() \} C.\text{wait}() \{ \text{SharedStateInv}() \wedge \text{Cond}(C) \}$$

$\text{Cond}(C)$  is the refinement of the shared state property denoted by condition  $C$ .

In our example:

- $\text{Cond}(\text{notzero}) = (N > 0)$
- $\text{Cond}(\text{notmax}) = (N < \text{MAX})$

# Partition shared state using conditions

---

```
/*@
predicate_ctor CCounter_shared_state (CCounter c) () =
    c.N |-> ?v &*& v >= 0 &*& c.MAX |-> ?m &*& m > 0 &*& v <= m;

predicate_ctor CCounter_nonzero (CCounter c) () =
    c.N |-> ?v &*& c.MAX |-> ?m &*& v > 0 &*& m > 0 &*& v <= m;

predicate_ctor CCounter_nonmax (CCounter c) () =
    c.N |-> ?v &*& c.MAX |-> ?m &*& v < m &*& m > 0 &*& v >= 0;

predicate CCounterInv(CCounter c) =
    c.mon |-> ?l
    &*& l != null
    &*& lck(l,1, CCounter_shared_state(c))
    &*& c.notzero |-> ?cc
    &*& cc != null
    &*& cond(cc, CCounter_shared_state(c), CCounter_nonzero(c))
    &*& c.notmax |-> ?cm
    &*& cm != null
    &*& cond(cm, CCounter_shared_state(c), CCounter_nonmax(c));
@*/
```

# ReentrantLock Conditions `await`

---

- The Condition method `await` makes a transformation from the generic shared state (`inv`) to the refined state corresponding to that condition (`acond`).

```
public interface Condition {  
  
    public void await();  
        //@ requires cond(this,?inv,?acond) &*& inv();  
        //@ ensures  cond(this, inv, acond) &*& acond();  
    ...  
}
```

# Partition shared state using conditions

---

- Conditions grant the access to the refined state condition

```
class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;
    Condition notzero;
    Condition notmax;
    void dec()
        //@ requires CCounterInv(this);
        //@ ensures CCounterInv(this);
    {
        mon.lock();
        //@ open CCounter_shared_state(this)();
        if (N == 0) notzero.await();
        //@ open CCounter_notzero(this)(); // refined state with N > 0
        N--;
        //@ close CCounter_shared_state(this)();
        mon.unlock();
    }
}
```

# Example (Bounded Counter)

---

- Conditions are defined with relation to both the representation invariant and the refined state.

```
class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;
    Condition notzero;
    Condition notmax;

    BCounter(int max)
        //@ requires max > 0;
        //@ ensures CCounterInv(this);
    {
        MAX = max;
        mon = new ReentrantLock();

        //@ close CCounter_shared_state(this);
        //@ close set_cond(CCounter_shared_state(this), CCounter_nonzero(this));
        notzero = mon.newCondition(); // notzero set to mean N > 0 !!

        //@ close set_cond(CCounter_shared_state(this), CCounter_nonmax(this));
        notmax = mon.newCondition(); // notmax set to mean N < MAX !!
    }
}
```

# ReentrantLock Conditions `await`

---

- Conditions are defined with relation to the representation invariant (`inv`) and the refined state (`pred`).

```
public class ReentrantLock {  
  
    ...  
    public Condition newCondition();  
    //@ requires lck(?t, 1, ?inv) &*& set_cond(inv, ?pred);  
    //@ ensures  lck(t, 1, inv)  &*& result != null &*& cond(result,inv,pred);  
}
```

# Partition shared state using conditions

---

- Different conditions give access to different refinements

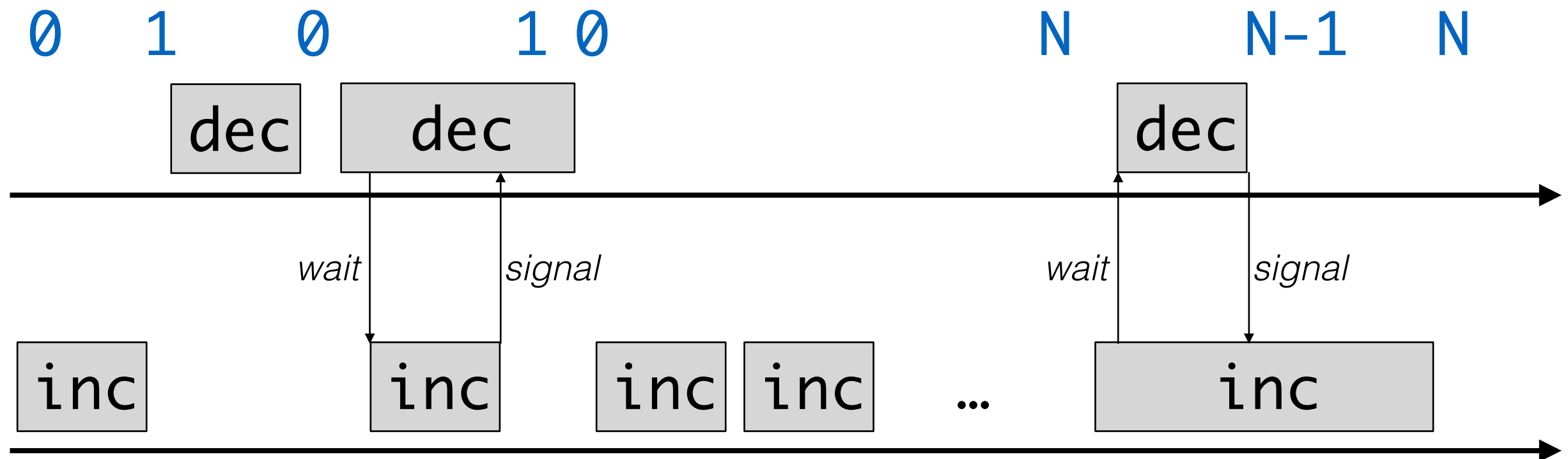
```
class CCounter {
    int N;
    int MAX;
    ReentrantLock mon;
    Condition notzero; Condition notmax;

    void inc()
    //@ requires CCounterInv(this);
    //@ ensures CCounterInv(this);
    {
        mon.enter();
        //@ open CCounter_shared_state(this)();
        if (N == MAX) notmax.await();
        //@ open CCounter_notmax(this)(); // refined state N < max
        N++;
        //@ close CCounter_shared_state(this)();
        mon.leave();
    }
}
```



# Wait and Signal work together

- Wait and signal represents a voluntary yielding of control flow and exclusive access to the monitor.
- Wait suspends a thread until a precondition is “satisfied”
- Signal states that a precondition is explicitly “satisfied”



# Ensure progress using signalling

---

- Wait and signal represents a voluntary yielding of control flow and exclusive access to the monitor.

```
void inc()
//@ requires CCounterInv(this);
//@ ensures CCounterInv(this);
{
    mon.enter();
    //@ open CCounter_shared_state(this)();
    if (N == MAX) notmax.await();
    //@ assert CCounter_notmax(this)();
    N++;
    //@ close CCounter_notzero(this)();
    notzero.signal();
    //@ close CCounter_shared_state(this)();
    mon.leave();
}
```

# Ensure progress using signalling

---

- Wait and signal represents a voluntary yielding of control flow and exclusive access to the monitor.

```
void dec()
//@ requires CCounterInv(this);
//@ ensures CCounterInv(this);
{
    mon.enter();
    //@ open CCounter_shared_state(this)();
    if (N == 0) notzero.await();
    //@ assert CCounter_notzero(this)();
    N--;
    //@ close CCounter_notmax(this)();
    notmax.signal();
    //@ close CCounter_shared_state(this)();
    mon.leave();
}
```

# Hoare Rule for *wait/signal* (*await/signal*)

---

$$\{ \text{SharedStateInv}() \} C.\text{wait}() \{ \text{SharedStateInv}() \wedge \text{Cond}(C) \}$$

$$\{ \text{SharedStateInv}() \wedge \text{Cond}(C) \} C.\text{signal}() \{ \text{SharedStateInv}() \}$$

**Cond(C)** is the refinement of the shared state property denoted by condition **C**.

In our example:

- **Cond(notzero)** =  $(N > 0)$
- **Cond(notmax)** =  $(N < \text{MAX})$

# Defending against unsound implementation

---

# Defending against unsound implementation

---

**Excerpt from Java API documentation:**

## **Implementation Considerations**

*When waiting upon a `Condition`, a "spurious wakeup" is permitted to occur, in general, as a concession to the underlying platform semantics.*

*This has little practical impact on most application programs as a `Condition` **should always be waited** upon in a loop, testing the state predicate that is being waited for.*

*An implementation is free to remove the possibility of spurious wakeups but it is recommended that applications programmers always assume that they can occur and so always wait in a loop.*

# Defending against unsound implementation

---

```
void inc()
//@ requires CCounterInv(this);
//@ ensures CCounterInv(this);
{
    mon.lock();
    //@ open CCounter_shared_state(this)();
    while(N==MAX) notmax.await();
    //@ assert CCounter_notmax(this)();
    N++;
    //@ close CCounter_notzero(this)();
    notzero.signal();
    //@ assert CCounter_shared_state(this)();
    mon.unlock();
}
```

# Defending against unsound implementation

---

```
void dec()
//@ requires CCounterInv(this);
//@ ensures CCounterInv(this);
{
    mon.lock();
    //@ open CCounter_shared_state(this)();
    while (N==0) notzero.await();
    //@ assert CCounter_notzero(this)();
    N--;
    //@ close CCounter_notmax(this)();
    notmax.signal();
    //@ assert CCounter_shared_state(this)();
    mon.unlock();
}
```



# Defending against unsound implementation

```
void inc()
  //@ requires CCounterInv(this);
  //@ ensures CCounterInv(this);
  {
    mon.lock();
    //@ open CCounter_shared_state(this)();
    while (N == MAX)
      /*@ invariant this.N l-> ?v &* & v >= 0
          &* & this.MAX l-> ?m
          &* & m > 0 &* & v <= m
          &* & this.notzero l-> ?cc &* & cc !=null
          &* & cond(cc, CCounter_shared_state(this), CCounter_nonzero(this))
          &* & this.notmax l-> ?cm
          &* & cm !=null &* & cond(cm, CCounter_shared_state(this), CCounter_nonmax(this));
      @*/
    {
      //@ close CCounter_shared_state(this)();
      try { notmax.await(); } catch (InterruptedException e) {}
      //@ open CCounter_nonmax(this)();
    }
    N++;
    //@ close CCounter_nonzero(this)();
    notzero.signal();
    mon.unlock();
    //@ close CCounterInv(this);
  }
```

# Summary

---

- Implementing preconditions using monitor conditions
- The operations await and signal yield control in a structured way.
- Unsound real implementations lead to active waiting code (that can also be verified) on the conditions.

Part V  
Construction of  
Concurrent ADTs

# Concurrent ADT Construction Steps

---

**Challenge:** Can we systematically transform and verify correct a “sequential” ADT implementation into an efficient “concurrent” ADT implementation?

# Concurrent ADT Recipe

---

1. Associate a monitor to the ADT (ReentrantLock - `mon`)
2. Define the **sequential** ADT Representation Invariant (**RepInv**) that talks about the shared state.

The Representation Invariant describes the memory footprint of the shared state, subject to other various conditions.
3. Define the **concurrent** ADT Representation Invariant that talks about the monitor and associated conditions.
4. In the implementation of each operation of the CADT:
  1. Get access to the shared state representation invariant, use `mon.lock()`
  2. When done and only if the shared state representation invariant holds, use `mon.unlock()`
5. Replace the ADT operation pre-conditions by monitor conditions inside the monitor (this part must be carefully thought!).
6. Design voluntary yielding points to implement the correct interleaving of operations.

# Concurrent ADT Construction Steps

---

To replace ADT operation pre-conditions by monitor conditions inside the monitor, we must consider the following aspects:

- When a thread enters a CADT operation and gets ownership of the representation invariant, it should check the state to satisfy (or not) the pre-condition (e.g., wants to decrement but counter value is zero)
- The thread must then await for the condition to hold (e.g, for the value to be  $> 0$ ).
- Conversely, whenever a thread running inside the CADT establishes any one of the monitor conditions (e.g., inc establishes value  $>0$ ), it has the duty to signal the condition (so that the runtime system may awake a waiting thread).
- Notice: signalling is there to help the system to progress, and simplify the implementation of monitors.